LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-PROC-210823

Obtaining Identiacal Results on Varying Numbers of Processors In Domain Decomposed particle Monte Carlo Simulations

N.A. Gentile, M.H. Kalos, T.A. Brunner

March 24, 2005

LLNL Computational Methods in Transport
Lake Tahoe, CA, United States
September 11, 2004 through September 16, 2004

Obtaining Identical Results On Varying Numbers Of Processors In Domain Decomposed Particle Monte Carlo Simulations

N. A. Gentile[1], Malvin Kalos[1] and Thomas A. Brunner[2]

[1] University of California, Lawrence Livermore National Laboratory*, Livermore California 94550 gentile1@llnl.gov,kalos1@llnl.gov
[2] Sandia national Laboratories**, Albuquerque New Mexico 87185 tabrunn@sandia.gov

Summary. Domain decomposed Monte Carlo codes, like other domain-decomposed codes, are difficult to debug. Domain decomposition is prone to error, and interactions between the domain decomposition code and the rest of the algorithm often produces subtle bugs. These bugs are particularly difficult to find in a Monte Carlo algorithm, in which the results have statistical noise. Variations in the results due to statistical noise can mask errors when comparing the results to other simulations or analytic results.

If a code can get the same result on one domain as on many, debugging the whole code is easier. This reproducibility property is also desirable when comparing results done on different numbers of processors and domains. We describe how reproducibility, to machine precision, is obtained on different numbers of domains in an Implicit Monte Carlo photonics code.

1 Description of the problem

There are two main issues that can cause a code to get different results when run on different numbers of domains. The first is that domain decomposition can cause the code to use a different sequence of pseudo-random numbers. The second, which also applies to deterministic codes which do not employ pseudo-random numbers, is that the order of operations on floating point numbers can change, leading to different results. We will examine both of these problems and describe solutions.

In the method we describe, problems are broken up into spatial domains. Computational work on each domain is performed by a single processor. Thus the number

* This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

** Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

of domains and the number of processors are the same. We have not considered having multiple processors work on a single domain, as could occur when threads are used.

Because we have done this work in the context of an Implicit Monte Carlo code, we will briefly describe that algorithm. The algorithm simulates the time-dependent interaction of photons and matter. It does this by creating, tracking, and destroying particles whose behavior models that of real photons in matter. This requires calculating probabilities for physical events such as emission and scattering. The behavior of each photon is determined by using a pseudo-random number to pick one of the behaviors. This is repeated until the photon is completely absorbed by the matter, leaves the domain, or reaches the end of the time step.

The behavior of matter is simulated on grid of zones, each with different material properties, such as different temperature and opacity. These zones lose energy when the emit particles, and gain energy when particles pass through them. Particles can visit a zone more than once (for example, by leaving and scattering back in from another zone.) In that case, a particle will deposit energy in the zone more than once.

Details of the algorithm can be found in [FC71]. This Implicit Monte Carlo program is used in the KULL [GKR98] and ALEGRA [BM04] inertial confinement fusion simulation codes. Parallel domain decomposition was accomplished using the algorithm described in [BUEG04].

Domain decomposition is necessary when the grid is too large to fit on the memory of one processor. It is also done to make problems run faster by bringing more computation resources to bear. To domain decompose the problem, we partition the grid and put parts on different processors. This necessitates moving particles between domains when they are tracked to domain boundaries. In order to debug this code, and have confidence in the results, we desire that a problem run on several domains (i.e., processors) get the same answer as when we run it on one domain (one processor).

The two issues that impact reproducibility are illustrated by considering the behavior of particles that cross a domain boundary. If the problem is run using one processor, there is (of necessity) one domain. With two or more processors, some zones will be on a domain boundary. Let Zone 1 and Zone 2 abut each other across a domain boundary, as shown in Fig. 1. Let Particle A be emitted in Zone 1 and enter Zone 2, scattering back into Zone 1. Let Particle B be emitted later in Zone 1 and stay there, executing a scatter.

When the problem is run on one processor, computations on Particle A continue until it is terminated (e.g., it leaves the problem through a transmitting boundary.) Only then are computations on Particle B begun. When it is run on two processors, Particle A is followed to the domain boundary and passed off to the processor on which Zone 2 resides. Then Particle B is followed until it is terminated.

Two things happen differently in the one domain case than in the two domain case.

First, the order of scatters of Particle A and Particle B is reversed. Scattering events use pseudo-random numbers to determine their outcomes (scattering angle, etc.) Hence the scatters will result in different behavior unless we ensure that the particles use the same pseudo-random numbers independent of the order in which those numbers are accessed.

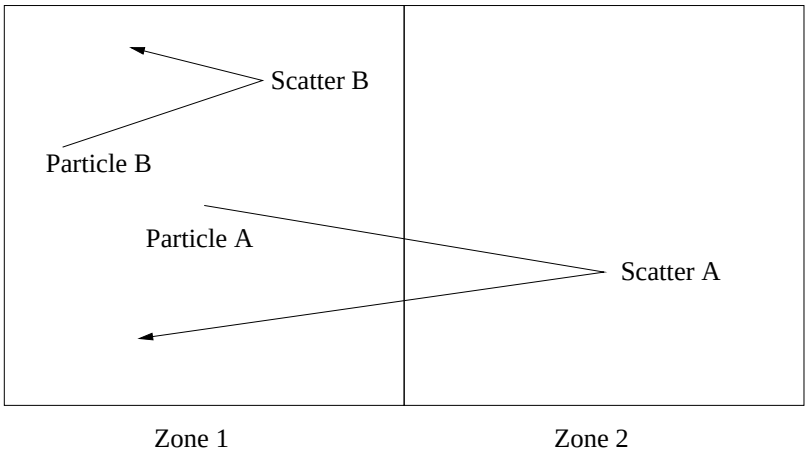

Fig. 1. Behavior of two particles in two zones of a Monte Carlo simulation. When both Zone 1 and Zone 2 are on the same processor, Particle A deposits energy on both legs of its path and executes a scatter before any calculations involving Particle B are done. When Zone 1 and Zone 2 are on different processors, Particle A deposits energy in Zone 1 and then leaves the domain. then Particle B deposits energy in Zone 1 and scatters. After Particle B is finished, Particle A reenters the zone and deposits energy. The different order of energy deposition and scattering (which uses a pseudo-random number) causes different results, unless the Monte Carlo algorithm is designed to eliminate the differences.

Second, the order of energy deposition in Zone 1 is different. Particle A deposits energy twice in Zone 1 before Particle B does when one processor is used. When two processors are used, Particle B deposits between the two deposition events involving Particle A. Addition in floating point arithmetic is not always exactly commutative: $(x + y) + z$ can differ from $x + (y + z)$ by a small amount on the order of roundoff. Hence, the final energy in Zone 1 may be slightly different in the two domain case than the one domain case. It might be thought that this small difference in results, on the order of roundoff, could be tolerated. We will demonstrate that it can have large effects on the result of a calculation by affecting the behavior of particles in subsequent time steps.

We will now discuss our solutions to these two issues.

2 Ensuring the invariance of the pseudo-random number stream employed by each particle

Domain decomposition alters the order in which particle event take place. The results for events which employ pseudo-random numbers will be different unless we ensure that each particle draws the same stream of pseudo-random numbers independent of the order in which it is simulated. The way we accomplish this is to give each particle its own pseudo-random number generator by giving each particle its own pseudo-random number generator state. An example will illustrate this.

A simple pseudo-random number generator is

$$s_n = a \cdot s_{n-1} + b \tag{1}$$

$$r_n = d \cdot s_n \tag{2}$$

Here a, b, and s are 64 bit unsigned integers, with $a = 2862933555777941757$, $b = 3037000493$, and s_n is the n^{th} state of the generator; d and r_n are double precision numbers, with $d = 5.4210108624275222 \cdot 10^{-20} \approx 1/2^{64}$ and r_n is the n^{th} pseudo-random number.

Applying the first part of this step maps the 64 bit unsigned integer s into another 64 bit unsigned integer. Since the maximum value of a 64 bit integer is 2^{64}, multiplying by the inverse of this number results in a value for r_n in the range $[0, 1]$.

In this pseudo-random number generator, s is referred to as the state. The current value of s (along with the constants a and b) completely determines the next value, and so it completely determines the subsequent stream of pseudo-random numbers.

If each particle has its own value of s, it will sample the same stream of pseudo-random numbers independent of the order of particle calculations. That is, for example, the fifth pseudo-random number used by Particle A will be the same, independent of which processor it is being simulated by, or how many other particles have been involved in computations since Particle A used its forth pseudo-random number.

Although we have illustrated the algorithm with a very simple pseudo-random number generator with a single integer as a state, it will work with more complicated pseudo-random number generators with larger states. The SPRNG library [SPRNG] has several pseudo-random number generators that work well in the context of this algorithm.

In order for this procedure to work, the first value of the state s, called the seed, will have to be determined in a manner that is independent of the domain decomposition. Its value will have to be determined from values that are invariant under domain decomposition. Some examples are global zone numbers, zone position, and the number of particles that have already been created in a given zone.

Using zone position is safer than using global zone numbers, because a code may not produce those. However, the position may not be invariant, because the code producing the grid positions may give slightly different (i.e., "jittery") positions with different numbers of domains. We will now describe an algorithm that gives invariant seeds from zone positions, provided that there is not too much jitter in these positions.

The first step is to find the minimum and maximum values, in each spatial dimension, of the locations of the zone centers on each processor. That is, we get the minimum and maximum values of x, y, and z on each domain. Then we find the minimum and maximum over all domains by using the MPI Allreduce command. Since these global values may have some jitter, we shave off the lower order bits. This is done by a "shaving" algorithm given in the appendix. Now we have three double precision numbers for the grid that are invariant over the number of domains.

Next, we loop over each zone in the grid and scale its position by the minimum and maximum values for the grid. That is, we calculate $(x_{\text{zone}} - x_{\min})/(x_{\max} - x_{\min})$, for x, y, and z. Because x_{zone} may also contain some jitter on different numbers of processors, we apply the shaving algorithm to these three numbers. This gives us

three numbers in $[0, 1]$ for each zone that are invariant over the number of domains. At least one of these numbers will be different for each zone. (Equality could occur for one or two of the numbers because, for example, zones in a one dimensional problem would have the same x and y values if it only had extend in the z direction.)

Then, we multiply these three numbers by $1.8446744 \cdot 10^{19} \approx 2^{64}$, which maps them into a 64 bit unsigned integers, and we add the three numbers together. This yields a 64 bit integer number that is different for each zone. To reduce correlations between zones, we then change this value into a new unique 64 bit unsigned integer by subjecting it to the DES hash algorithm [PTVF02]. This yields one 64 bit unsigned integer for every zone that is invariant over the number of domains. To get initial seeds for each particle in the zone, we increment the zone value by one and apply the DES hashing. This gives each particle a unique seed that is independent of the number of domains. Thus each particle accesses the same pseudo-random number stream, independent of the number of domains.

3 Ensuring that addition is commutative

Using the algorithm described above, we can ensure that all particles access the same pseudo-random number stream independent of the order in which they are simulated. They will still, however, deposit energy in the zones in a different order when the number of domains is changed. Because floating point addition is not exactly commutative, there will small differences in the total energy deposited in each zone at the end of the time step.

These differences are on the order of roundoff. However, we cannot tolerate them because they will eventually have macroscopic consequences. This is because the energy deposited in the zone will affect the temperature and opacity of the zone in the next time step. The opacity can be a nonlinear function of the temperature, and so small differences in the temperature can be magnified. Differences in the value of the opacity will cause differences in the deposition of energy by every photon that enters the zone. This in turn will effect the creation and behavior of particles in the zone. Eventually, some particle will behave differently (e.g., not scatter) because of slight differences in the values of temperature and opacity. This different behavior will have a macroscopic effect on the problem, which will affect other particles in subsequent time steps. Soon, the difference between the two cases will be as large as if different random number streams were used.

This effect is illustrated in Fig. 2. This plot shows the difference in the temperature in the first zone of three different simulations of a test problem from [FC71]. The opacity is given by equation 5.2 in [FC71]:

$$\sigma = \frac{27}{\nu^3 T^{3/2}}(1 - \mathrm{e}^{\nu/T})\frac{1}{\mathrm{cm}} \tag{3}$$

with ν and T in keV

The simulations used 100 zones with $\Delta x = 0.4$. It was run for 400 time steps with Δt fixed at $2.0 \cdot 10^{-12}$ sec. A temperature source with $T_s = 1.0$ keV was applied at $x = 0$. The initial temperature was 0.01 keV. The equation of state had a constant heat capacity of $C_v = 8.11829 \cdot 10^9$ erg/(cm^{3}keV) $= 0.5917 a T_s^3$, where a is the radiation constant. Each simulation used 1000 particles in each time step.

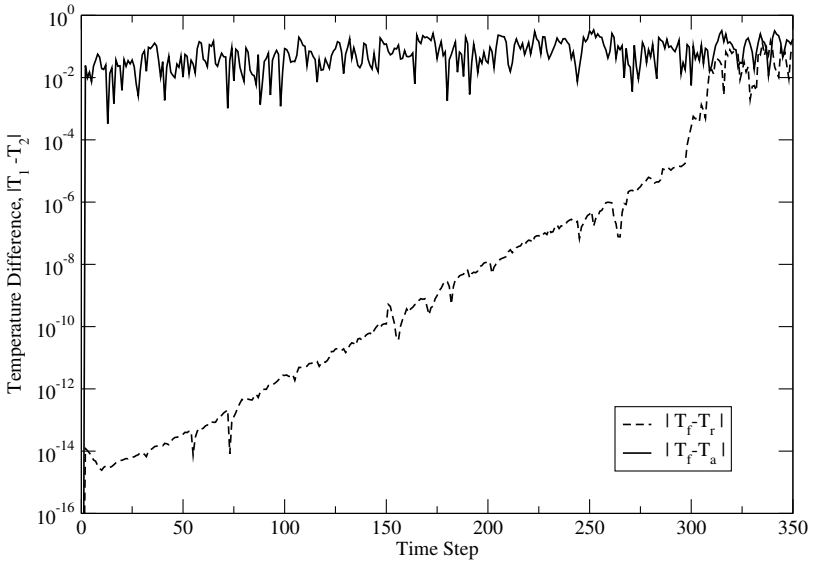

Fig. 2. The absolute value of the difference in temperature in the first zone of three simulations vs. time step. Two simulations used the same pseudo-random number stream but different particle order. (These are denoted the "forward", T_f, and "reverse", T_r, simulations.) The third simulation used a different pseudo-random number stream (the "alternate", T_a, simulation.) $|T_f - T_r|$ is depicted with a solid line and $|T_f - T_a|$ with a dotted line. $|T_f - T_a|$ is large at the beginning, and fluctuates but does not increase with time. $|T_f - T_r|$ begins at a value near the roundoff error for double precision arithmetic, but grows and eventually jumps to same level as the difference between the "forward" and "alternate" simulations.

All three simulations were run on one processor, and all used the pseudo-random number described above. Two simulation employed the algorithm described above which ensured that particles got the same pseudo-random number stream in each case. The only difference between these simulations was that the particles were run in a different order. After new particles were created at the beginning of each time step, the order of the list was reversed before they were tracked. Thus any difference in the results of these two simulations is due to differences in the order of floating point addition when the particles deposit energy.

The third simulation used a different pseudo-random number stream entirely. This was accomplished by adding 78654092354 to the initial value of s in each zone in the simulation.

Fig. 2 plots $|T_f - T_r|$ and $|T_f - T_a|$ versus the time step in the simulation. Here T_f is the temperature in the first zone when the particles are run in the usual order, T_r is the temperature in that zone when the particle order was reversed, and T_a is the temperature in the run which used a different pseudo-random number stream.

The difference between runs using different pseudo-random number streams is fluctuates between 0.01 and 0.1 throughout the simulation. (The value of the temperature in the first zone quickly becomes approximately equal to the $T_s = 1.0$.) The

difference between runs using the same pseudo-random number stream is initially very small. The difference in the first time step is $O(10^{-14})$, which is the size of roundoff errors. This causes a difference in the opacity in the zone, which means that photons in the zone will deposit slightly different amounts of energy in the two simulations. The difference in temperature grows with each time step. However, the difference remains small, and global measures like the total number of particles simulated remain the same until time step 304.

At time step 304, the accumulated difference in temperature is large enough to change the large-scale behavior of the code. A different number of particles are created in the two simulations in this time step. After that, the differences quickly grow until they are of the same order as the difference we find compared to the simulation with a completely different pseudo-random number stream.

The cure for this problem is relatively simple. Floating point addition is not commutative, but integer addition is. We eliminate differences in results by mapping the energy to a 64 bit unsigned integer before doing the addition.

This scaling is a simple multiplication that maps the range of photon energy in the problem into the range that a 64 bit unsigned integer can hold, which is $[0, 2^{64} - 1]$. This number changes with each time step, and is calculated at the beginning of each time step. The multiplier is

$$S = 2^{63}/(E_{\text{census}} + E_{\text{source}}). \tag{4}$$

Here E_{census} is the sum of the energy of photons present at the beginning of the time step, and E_{source} is the total amount of energy added to photons at the beginning of the time step (e.g., $aT^4V_{\text{zone}}\Delta t$ for the thermal radiation emitted from a zone.) We have used 2^{63} rather than $2^{64} - 1$, the maximum value of an unsigned 64 bit integer, as a safety factor. Note that S is a double precision value, not an integer.

To calculate the energy deposited into a zone, we sum the 64 bit unsigned integers obtained from scaling the energy deposited by each photon:

$$E^{\text{int}}_{\text{photon}} = \text{integer}(E_{\text{photon}} \times S + 0.5) \tag{5}$$

$$E^{\text{int}}_{\text{dep}} = E^{\text{int}}_{\text{dep}} + E^{\text{int}}_{\text{photon}} \tag{6}$$

Here $E^{\text{int}}_{\text{dep}}$ is an unsigned 64 bit integer, and integer() represents a cast from a double precision value to a 64 bit unsigned integer. The 0.5 is added to round to the closest integer instead of simply truncating the fractional part of the value.

At the end of the time step, when deposition is complete, we get the total energy deposited in each zone by reversing the multiplication:

$$E_{\text{dep}} = \text{double}(E^{\text{int}}_{\text{dep}}/S) \tag{7}$$

Here, E_{dep} is of type double precision, and double() represents a cast from a 64 bit unsigned integer to double precision.

4 Results

Here we demonstrate that the algorithm outlined above will give the same results for simulations using different numbers of processors. The results shown are for the analytic transport benchmark of Su and Olson [SO97].

This test problem has an initially cold slab with constant opacity which is heated by an isotropic source of photons. Interactions with the radiation field heat the matter and cause it to eventually reach thermal equilibrium. The results depicted in Fig. 3 are for the case $\kappa_a = 1, \kappa_s = 0$, at $\tau = 0.3$, in the notation of [SO97].

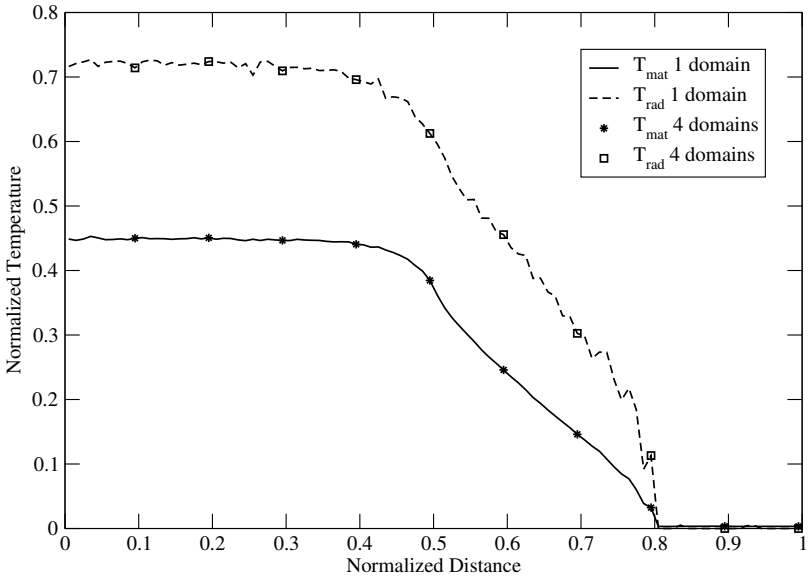

Fig. 3. Matter and radiation temperature in the Su Olson test problem vs. spatial coordinate. Solid lines depict the results of a simulation run on one domain. Symbols depict the results for every tenth zone of a simulation run on four domains. The four domain results are identical to the one domain results, even reproducing the details of statistical fluctuations.

Fig. 3 shows results for matter and radiation temperature for one and four domain runs. Simulations using one domain are solid lines. Every tenth point for the four domain runs is plotted with a symbol. The four domain results exactly reproduce the one domain results. Even the statistical noise in the Monte Carlo solution is reproduced.

As part of the KULL regression suite, this test is run weekly. The results for the temperature in every zone for one, two, and four domain runs are printed to sixteen places, and the files are compared to ensure that the results are identical. This reproducibility has been demonstrated for larger problems, up to 1024 processors.

5 Conclusions

We have described algorithms that can be used to make domain decomposed Monte Carlo photonics code produce the same answer, bit for bit, on various numbers of domains.

Two main issues are introduced when the number of domains can vary. The first is that the particles may not get the same pseudo-random number stream. The second is that the order of operations can change, causing differences in the results arising from the fact that floating point addition is not commutative.

The first issue is eliminated with by giving each particle its own random number generator state, and seeding these states in a manner that is independent of the number of domains. The second issue is eliminated by using integers, which are commutative, to do addition.

We have demonstrated that we can achieve bit for bit agreement on problems when the number of domains varies from one to one thousand.

Appendix: shave algorithm

This algorithm takes a double precision number x and truncates all but the highest N_d base-ten digits. We use $N_d = 7$ in our application.

shave(x, N_d):
  if $x == 0.0$:
    return 0.0
  Store the magnitude and sign of x.
  $x_{\text{sign}} = \text{sign}(x)$
  $x_{\text{abs}} = \text{abs}(x)$
  Truncate base-ten exponent to get magnitude of x.
  integer $n = \text{integer}(\log_{10}(x_{\text{abs}}))$
  Scale x_{abs} to be between 10^{N_d} and $10^{N_{d+1}}$.
  double $s = 10^{n-N_d}$
  double $x_{\text{scaled}} = x_{\text{abs}}/s$
  Shave off digits by casting to integer.
  integer $x_{\text{scaled}}^{\text{shaved}} = \text{integer}(x_{\text{scaled}})$
  Restore correct magnitude and sign.
  return $s \cdot x_{\text{sign}} \cdot x_{\text{scaled}}^{\text{shaved}}$

References

[FC71] Fleck, Jr., J. A., Cummings, J. D. : An Implicit Monte Carlo Scheme for Calculating time and frequency dependent nonlinear radiation transport. J. Comp Phys., **31**, 313–342 (1971)

[GKR98] Gentile, N. A., Keen, N., Rathkopf, J.: The KULL IMC Package, Tech. Rep. UCRL-JC-132743, Lawrence Livermore National Laboratory, Livermore CA (1998)

[BM04] Brunner, T. A., Mehlhorn, T. A.: A User's Guide to Radiation Transport in ALEGRA-HEDPP, Tech. Rep. SAND-2004-5799, Sandia National Laboratories, Albuquerque, NM (2004)

[BUEG04] Brunner, T. A., Urbatsch, T. J., Evans, T. M., Gentile, N. A., Comparison of Four Parallel Algorithms for Domain Decomposed Implict Monte Carlo, Tech. Rep. SAND-2004-6694J (2004) Submitted to J. Comp. Phys.

[SPRNG] http://sprng.cs.fsu.edu/

[PTVF02] Press, W. Tuekolsky, S.A., Vetterling, W. T., Flannery, B. P.: Numerical Recipies in C++. Cambridge University Press, United Kingdom (2002)

[SO97] Su, B., Olson, G. L.: An analytical benchmark for non-equilibrium radiative transfer in an isotropically scattering medium. Ann. Nucl. Energy, **24**, No. 13 (1035–1055 (1997)